Variables and constants

One of the responsibilities of a programmer is to ensure the program is *readable and modifiable by other people*, in addition of making the program valid and readable for the computer. This is called *maintainability*.

To make maintainable programs, there are several general guidelines you should follow. Two of the most important are:

- follow the *"Don't-Repeat-Yourself principle" (DRY)*: whenever you need to refer two or more times to the same concept, only write it once, give it a name, and reuse that name in multiple places. This is to reduce the number of potential errors that would happen if the concept needs to be changed in the future.
- follow the *No Magic Values rule*: do not place numeric values that have a special meaning in the business/user domain in the middle of program logic; instead, give them a name at the beginning and document their role in program comments. This is to ensure that the reader of the code can understand the business/user domain before he/she starts reading the rest of the program.

For example, say that you want to decide a price for a train ticket; the normal price is 30EUR; people under 6 have a reduced "children" fare of 5EUR; people between 6 and 25 get a "student" fare of 20EUR, over 65 get a "senior" reduction of 25%. A naive encoding of this logic could be:

```
price = 30;
if (age <= 6)
   price = 5;
else if (age > 6 && age <= 25)
   price = 20;
else if (age >= 65)
   price = price * 1.25;
```

However there are two problems here. One is that this code violates the DRY principle: the ages "6" and "25" are both repeated two times. The other is that it violates the "no magic values" rule: each age and price value are business values which cannot be interpreted by just looking at the code.

Factoring and refactoring, factoring constants

When you see two or more instances of the same thing, you should immediately think "DRY principle" and attempt to group them together with a single name. Likewise, if you see a numeric value whose significance is dependent on the larger (human) context, you should give it a name and explain it early.

This process of grouping things and giving them an isolated name and explanation is called *factoring* (for the 1st time) and *refactoring* (from the 2nd time onward).

Usually (re)factoring is explained/performed in 3 forms:

- factoring numeric values towards the beginning of a program: *constant factoring*;
- grouping/moving statements within a small context: *local refactoring*;
- moving/restructuring larger portions of a program from one place to another: *global refactoring*.

Constant factoring and constant naming convention

For the small example above, only constant factoring is sufficient to fix the two problems. So we need to place a single declaration at the beginning for all the special values.

For this three steps are necessary:

1. choosing a name;
2. moving the value at the beginning and bind it to the name;
3. rewriting the place where the value was earlier expressed, by its new name.

The process of *choosing a good name* thus forms an essential part of constant factoring. There are many ways to do this, but the entire software engineering industry has settled on a general consensus: *when factoring constants, it is customary to give them a name all in capitals, with words separated by underscores, "LIKE_THIS"*.

A rule that determines how to name things is called a *naming convention*; this specific rule above is usually phrased as follows: "the naming convention for constants is to use capitals, with words separated by underscores".

Note this is not strictly necessary from the computer's perspective, but people who read other people's code usually learn to expect constants named this way, so you should do the same too.

For example:

```
int CHILD_AGE = 6;
int YOUNG_AGE = 25;
int SENIOR_AGE = 65;
double BASE_FARE = 30;
double YOUNG_FARE = 20;
double CHILD_FARE = 5;
double SENIOR_REDUCED_FARE = BASE_FARE * 1.25;


price = BASE_FARE;
if (age <= CHILD_AGE)
   price = CHILD_FARE;
else if (age > CHILD_AGE && age <= YOUNG_AGE)
   price = YOUNG_FARE;
else if (age >= SENIOR_AGE)
   price = SENIOR_FARE;
```

Constants vs. mutability, using `final` in Java

One new problem introduced by factoring constants is that if you use a "regular" variable to store the numeric value, nothing prevents a programmer from (erronously or maliciously) *changing* this value over time. So a constant may end up not being really constant after all, and troubleshooting code that changes constants becomes very difficult.

To avoid this, every imperative language offers a special construct to declare that a constant is just that, *immutable*: the program cannot change their value, and any attempt to do so would trigger a compile-time error.

In Java, the way to do that is to write "`final`" in the declaration, like this:

```
final int CHILD_AGE = 6;
final int YOUNG_AGE = 25;
final int SENIOR_AGE = 65;
final double BASE_FARE = 30;
final double YOUNG_FARE = 20;
final double CHILD_FARE = 5;
final double SENIOR_REDUCED_FARE = BASE_FARE * 1.25;
```

By doing this, a statement of the form "`SENIOR_AGE = 42`" later in the program becomes invalid.

Important concepts

- *maintainability*
- *DRY principle*
- *No Magic Values rule*
- *factoring* and *refactoring*
- *naming convention*
- *naming convention in Java for constants*
- *mutability* and `final` in Java

Further reading

- "DRY principle" on Wikipedia and other sources
- Programming in Java, section 4.7.2 (pp. 177-180)
- Absolute Java, section 1.4 (pp. 46-48)

Copyright and licensing